Formalization and Abstract Implementation of Rewriting with Nested Rules*

Sergio Antoy and Stephen Johnson
Computer Science Department
Portland State University
P.O. Box 751, Portland, OR 97207, U.S.A.
{antoy,stephenj}@cs.pdx.edu

ABSTRACT

This paper formalizes term rewriting systems (TRSs), called *scoped*, in which a rewrite rule can be nested within another rewrite rule. The right-hand side and/or the condition of a nested rule can refer to any variable in the left-hand side of a nesting rule. Nesting of rewrite rules is intended to define a lexical scope with static binding. Our work is applicable to programming languages in which programs are modeled by TRSs and computations are executed by rewriting or narrowing. In particular, we consider a class of non-confluent and non-terminating TRSs well suited for modeling modern functional logic programs. We describe an abstract implementation of rewriting and narrowing for scoped TRSs to show that scopes can be easily handled irrespective of the evaluation strategy. The efficiency of rewriting within a scoped TRS, measured using a narrowing virtual machine, is comparable to the efficiency of rewriting for non-scoped TRSs.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Multiparadigm Languages—*Functional Logic Languages*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages; F.4.2 [**Mathematical Logic and Formal Languages**]: Grammars and Other Rewriting Systems—*Lexically Scoped Rewriting Systems*

General Terms

Design, Languages, Theory

Keywords

Term Rewriting Systems, Narrowing, Block Structured, Functional Logic Programming, Non-Determinism

*This research has been supported by the NSF grants CCR-0218224 and CCR-0110496.

PPDP'04, August 24–26, 2004, Verona, Italy.

1. INTRODUCTION

Term rewriting systems [4, 5, 10, 16] are versatile models of computation in their own right and underlie several modern declarative programming languages, e.g., ASF+SDF [19], Elan [6] and Maude [8] to name a few. In particular, programs in the functional logic programming languages Curry [12] and Toy [17] are generally regarded as TRSs and are executed by narrowing. Modeling these programs as TRSs is somewhat necessary and convenient. The lambda calculus and many of its variations and extensions are inappropriate models for functional logic programming languages even though many declarative programming languages have a strong functional flavor. These languages support the definition of "non-deterministic operations" including constants seen as nullary symbols. For example, consider the following Curry program:

```
coin = 0
coin = 1
```
(1)

The expression `coin` evaluates non-deterministically to either `0` or `1`. Furthermore, functional logic languages are executed by *narrowing*. Narrowing is a computation that evaluates functional-like expressions possibly containing uninstantiated logic variables. Narrowing supports both terse code and expressive abstractions. This is illustrated in Display (2) with `last`, which is a function that computes the last element of list.

```
[] ++ y = y
(x:xs) ++ y = x:(xs++y)
last l | l =:= x++[e] = e
    where x, e free
```
(2)

The operation "`++`" is the usual list concatenation. The operation `last` is defined by a conditional rewrite rule. The condition, `y=:=x++[e]`, is an equational constraints containing two uninstantiated variables `x` and `e`. Narrowing computes values for these variables that satisfy the constraint and thus determines the last element of a list.

Functional logic languages are often seen as extensions of functional languages. However, even though the lambda calculus is the traditional underlying semantic and operational model of functional languages it is inadequate for functional logic languages because of non-determinism and narrowing. A TRS is a primary semantic and operational model for a functional logic program. A TRS is also a convenient model because several strategies discovered in the last decade, see [2] for a summary, become available for the execution of programs in these languages.

Since functional logic languages are often seen as extensions of functional languages, a majority of semantic concepts and syntactic constructs of functional languages are provided with little or no change in functional logic languages. As a case in point, Curry has been designed as an extension of Haskell. In this spirit, the definition of a function—or more precisely a symbol—can be nested within the definition of another function. However, in Curry these symbols may include non-deterministic operations of the kind shown earlier, therefore what is really being nested is not a function, but a set of rewrite rules because the semantics of the language models programs as TRSs. No rigorous semantics exists for the nesting construct in functional logic languages.

Nesting is intended to define a lexical scope which provides a name space for both signature symbols and variables. This paper focuses only on the variables. In a nested rewrite rule, the right-hand side and/or the condition can refer to a variable defined by the left-hand side of a nesting rewrite rule. This variable is referred to as *non-local*. In the following Curry program,

```
reverse []     = []
reverse (x:xs) = appelem (reverse xs)
  where appelem []     = [x]
        appelem (y:ys) = y:appelem ys
```
(3)

both rewrite rules defining the function `appelem` are nested within the second rule of the function `reverse`. Nesting is syntactically introduced by a *where block* declaration together with appropriate indentation known as *off-side* rules. The right-hand side of the first rule of `appelem` contains a variable, `x`, which does not occur in the left-hand side. The intended meaning is that this occurrence is bound to the variable with the same identifier introduced by the left-hand side of the nesting rule, the second rule of `reverse`. The above program would be an ordinary TRS except for the occurrence of a non-local variable.

Nesting in functional programs is syntactically similar to the above examples. In fact, in Haskell it is identical. Compilers remove nesting typically with transformations such as *lambda-lifting* [15] or *closure conversion*. Lambda-lifting changes a non-local variable into a local one by explicitly passing it to nested functions that directly or through other nested functions refer to it. If all the non-local variables are explicitly passed to nested functions, nesting becomes irrelevant, except for potential name clashes that can be removed by renaming, and nested functions become equivalent to top-level functions. The result of lambda-lifting the previous example follows:

```
reverse []     = []
reverse (x:xs) = appelem x (reverse xs)
appelem z []     = [z]
appelem z (y:ys) = y:appelem z ys
```
(4)

For ease of reading, we have preserved the original identifiers although operations with the same identifiers differ in the two programs. In particular, `appelem` takes one extra argument, the first one, and is no longer nested within `reverse`.

Closure conversion is a lower-level compilation technique for the execution of functions, or more generally procedures in a variety of languages, with free variables. Functions are compiled into a pair consisting of pure code and a mapping referred to as the environment, which defines bindings for the free variables in the code. Several implementations, e.g., the use of machine registers, hierarchical incremental updates, etc., have been proposed for this structure.

Lambda-lifting and closure conversion are compilation techniques that have not been well defined in the presence of narrowing and non-determinism. Therefore, the application of either lambda-lifting or closure conversion to functional logic programs is at the very least unsatisfactory since functional logic programs are executed by narrowing and may define non-deterministic operations. This paper formalizes a notion of scoped TRSs, a TRS with nested rewrite rules, and proposes an abstract implementation of rewriting and narrowing for a scoped TRS. On the conceptual level, our work provides a rigorous semantics for a feature commonly found in programs. This semantics is natural and direct rather than obtained by a program transformation. On the practical level, our semantics has some similarities with closure conversion and can be the basis of an implementation. This implementation is simple, but non-trivial, since it must be independent of the rewriting strategy. We also show that a realistic implementation can be competitive with lambda-lifting in some cases.

In Section 2 we introduce background information and the notation necessary to understand our work. In Section 3 we describe an implementation of rewriting. This implementation is abstract in the sense that it avoids details that may be important for a real implementation, but that are irrelevant to our presentation. By remaining *abstract* we hope to ease the understanding of our idea and to focus on the key issues. In Section 4 we formalize scoped TRSs and we present how to execute rewriting and narrowing computations for TRSs with nested rewrite rules. Our presentation is abstract in the sense discussed earlier and for the same reasons. In Section 5 we summarize the differences in execution time and memory consumption of a set of rewriting and narrowing computations. Our benchmark is obtained using a virtual machine for narrowing computations [3] which is intended to be efficient. This summary shows that there is a modest cost to provide the infrastructure necessary to compute with non-local variables. It also shows that computations in which non-local variables are frequently accessed generally execute faster. In Section 7 we offer our conclusion.

2. PRELIMINARIES

This section briefly recalls basic notions of term rewriting [4, 5, 10, 16].

A *term rewriting system* is a pair $\langle R, \Sigma \rangle$, where R is a set of *rewrite rules* and Σ is a signature. The signature is a set of *symbols*, where any symbol f has an *arity*, a non-negative integer that defines the number of arguments of an application of f. The set of *terms* constructed over Σ and a countably infinite set $\mathcal{X}$ of *variables*, $\textsc{Term}(\Sigma \cup \mathcal{X})$, is defined as follows: every variable is a term; if f is a symbol of arity $n \geqslant 0$ and $t_1, \ldots, t_n$ are terms, $f(t_1, \ldots, t_n)$ is a term. $\mathcal{V}ar(t)$ is the set of variables occurring in a term t. An *occurrence* or *position* of a term t is a sequence of positive integers, $\langle p_1, \ldots, p_k \rangle$, $k \geqslant 0$, identifying a subterm u in a term t as follows: if $k = 0$ then $u = t$, otherwise if $t = f(t_1, \ldots, t_n)$ then u is the occurrence of t_{p_1} at $\langle p_2, \ldots, p_k \rangle$. The expression $t|_p$ denotes the subterm of t at p. The expression $t[u]_p$ denotes the term obtained from t by replacing $t|_p$ with u.

A *substitution* is a mapping from variables to terms. A substitution σ is extended to terms by $\sigma(f(t_1, \ldots, t_n)) =$

$f(\sigma(t_1)\ldots,\sigma(t_n))$. A *rewrite rule* is a pair $l \to r$, where l is a non-variable term and r is a term such that $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$. A TRS defines a rewrite relation on terms as follows: $t \to_{p,l\to r,\sigma} u$ if there exists a position p in t, a rewrite rule $l \to r$ and a substitution σ with $t|_p = \sigma(l)$ and $u = t[\sigma(r)]_p$. An instance of a left-hand side, $\sigma(l)$, is called a *redex* (*red*ucible *ex*pression). The corresponding instance of the right-hand side, $\sigma(r)$, is called the redex *replacement* and the term u is called a *reduct* of t.

A computation is a sequence of rewrite steps. Terms whose subterms are not redexes are *normal forms*. Several distinct redexes may occur in a term. A TRS does not specify which redex to replace. This choice is the task of a *strategy*. Strategies play a crucial role in the execution of programs modeled by TRSs. Our formalization and implementation of rewriting in scoped TRSs is independent of any strategy and therefore accommodates any strategy that a language implementation deems most appropriate.

3. ABSTRACT IMPLEMENTATION

This section presents an abstract implementation of rewriting. There are no new concepts in this material. Our overall goal is to show that the formalization of scoped rewriting presented in the next section can be implemented naturally, directly, and with acceptable performance.

Efficient implementations of rewriting are intertwined with a strategy. The strategy depends on the class of TRSs considered, e.g., strongly sequential, constructor based, weakly orthogonal, etc. The concepts are highly technical and the details are fairly complicated. This machinery, intended to optimize the selection of a redex, is irrelevant to our presentation. Therefore, we discuss an *abstract* implementation of rewriting. The merit of this implementation is to leave out both the strategy and its many details. This implementation would be naive for practical purposes. Later, we will revisit this implementation and show that it can be extended to scoped TRSs with a few well-localized changes. We believe that this form of presentation will help understanding the differences between ordinary and scoped rewriting and will guide, as it did in our case, more realistic implementations [3].

Rewriting can be implemented with very little machinery when the efficiency of computing a normal form is not a major concern. Two components are crucial to any such implementation: a suitable type for the *representation* of terms and a function that executes a *rewrite step*. We limit our discussion to *left-linear* TRSs, i.e., TRSs in which a variable may occur at most once in the left-hand side of a rewrite rule. Our technique is indifferent to left-linearity, but this assumption simplifies our implementation of abstract rewriting. The function *match*, described later, matches the left-hand side of rule against a term. If the left-hand side is linear, this function can bind a variable occurring in the left-hand side without checking for previous bindings. Furthermore, TRSs modeling programs in many functional and functional logic programming languages are left-linear. We describe an abstract implementation of rewriting in the sense that we ignore (by encapsulation) many program details, any efficient pattern matching [13], or any efficient strategy [2]. In particular, we hide the details of the representation of terms.

Terms are trees. Hence they are naturally and easily represented as dynamic linked structures in imperative programming languages and as algebraic data types in declarative languages. Here we assume the existence of a type *Term*. An instance or object of a non-variable term has two components: a *root* and a sequence of zero or more *arguments*. The root abstracts a symbol of the TRS's signature. A *symbol* has attributes such as a name, an arity, etc. The arguments are terms themselves. For example, referring to Display (4), let t be the term `reverse []`. The root of t is the symbol `reverse`. t has only 1 argument. The first and only argument of t is the term `[]`. *Term* is an overloaded abstract function that takes a symbol and its arguments and constructs the corresponding term. The functions *root* and *arg* decompose a term and are related to *Term* as follows:

$$\begin{array}{lcll} \mathit{root}(\mathit{Term}(r, a_1, \ldots, a_n)) & = & r & n \geqslant 0 \\ \mathit{arg}(i, \mathit{Term}(r, a_1, \ldots, a_n)) & = & a_i & 1 \leqslant i \leqslant n \end{array} \quad (5)$$

Variables are terms, too. We are not interested in the typical attributes of a variable, such as its identifier or its binding, but in the fact that if a variable occurs in the left-hand side of a rewrite rule, then our assumption of left-linearity ensures that its occurrence is unique. Therefore, an instance or object of a variable term has a single component: its unique *occurrence* in the left-hand side of the rewrite rule that defines it. The type *Occurrence* abstracts a sequence of positive integers that identify a subterm of a term. For example, the occurrence of `ys` in the left-hand side of the last rewrite rule of Display (4) is $\langle 2, 2\rangle$ since `ys` is the second argument of the second argument of the left-hand side. *Var* is an overloaded abstract function that takes a sequence of positive integers and constructs a variable. The functions *occur*, *first* and *rest* are related as follows:

$$\begin{array}{lcll} \mathit{occur}(\mathit{Var}(p_1, \ldots, p_n)) & = & (p_1, \ldots, p_n) & n \geqslant 1 \\ \mathit{first}(p_1, \ldots, p_n) & = & p_1 & n \geqslant 1 \\ \mathit{rest}(p_1, \ldots, p_n) & = & (p_2, \ldots, p_n) & n \geqslant 1 \end{array} \quad (6)$$

Rewriting is implemented as a sequence of rewrite steps. A rewrite step of a term is accomplished with two key simple operations: (1) find a redex and (2) compute its replacement. The following function *match* tells whether a rewrite rule $l \to r$ is applicable to a term t. Left-linearity simplifies this computation. The term t can be thought of as a subterm of some larger expression evaluated during the execution of a program in some programming language. The selection of t in the larger expression is made by a strategy, a crucial conceptual and practical component of the language's implementation. Our technique is applicable irrespective of the strategy, therefore we ignore strategy related issues.

```
boolean match(Term t, Term l) {
    if isVariable(l) return true;
    else {
        int n = arity(root(l));
        return root(t) = root(l)
            and match(arg(1,t),arg(1,l))
            and ···
            and match(arg(n,t),arg(n,l));
    }
}
```

The following function, *replace*, constructs the replacement of a redex. Suppose that t is a term and $l \to r$ is a rewrite rule such that $\sigma(l) = t$, for some substitution σ. The replacement of t is $\sigma(r)$. The substitution σ is not explicitly

computed by *match* and it is not explicitly required to compute $\sigma(r)$. Left-linearity simplifies this computation. The variables in l and r are identified by their occurrence in l. In the initial call to *replace* the arguments are t and r. A subterm of r is passed down to recursive calls. The value returned by *replace* is $\sigma(r)$.

```
Term replace(Term t, Term r) {
  Term subterm(Term t, Occurrence o) {
    if isEmpty(o) return t;
    else return subterm(arg(first(o),t),rest(o));
  }
  if isVariable(r) return subterm(t, occur(r));
  else {
    int n = arity(root(r));
    return Term(root(r),replace(t,arg(1,r)), ···
                        replace(t,arg(n,r)));
  }
}
```

The functions *match* and *replace* are the essential blocks of an abstract implementation of rewriting. Let t be a term to rewrite. Some strategy, using *match*, finds a redex s in t and a rewrite rule applicable to s. The function *replace* computes the replacement s' of s. The reduct t' of t is t with s substituted by s'. If the strategy is complete (normalizing), by repeating the process to t' until no more redexes are found, one obtains the normal form of t. The details that we omit depend on the strategy, which is irrelevant to our technique, and the programming language of the implementation.

The abstract implementation of rewriting presented in this section is useful to understand the crucial problem of rewriting with nested rewrite rules. In the execution of *replace*, if the argument r were a non-local variable the term bound to r would not be a subterm of t. Hence, the expression *subterm(t,occur(r))* would either produce an unintended result or be undefined. This observation also suggests how to correct this problem. This will be presented after the formalization of scoped TRSs presented in the next section.

4. FORMALIZATION

In this section we formalize a notion of a TRS with nested rewrite rules. We call these TRSs *scoped*. We extend the abstract implementation of rewriting for ordinary TRSs presented in the previous section to scoped TRSs. As we already pointed out, this implementation is missing information to find the bindings of non-local variables. These bindings are created (implicitly in our abstract implementation) when a term is matched against the left-hand side of a rewrite rule. Therefore, loosely speaking, all that is required to rewrite with nested rewrite rules is to "keep around" these bindings. Since we represent variables by their occurrences, it suffices and it is quite convenient to access the term matching the left-hand side of the rule that defines the binding of a variable.

This apparent conceptual simplicity should not misled the reader to think that the problem we face is trivial. The notion of a scope, and the consequent need to manage a context or environment, are as old as the early programming languages. In the next section we will briefly recall a few non-trivial approaches proposed to solve these problems. We remark that a specific issue of our discussion is that we make no a priori assumption on the evaluation strategy, but we expect any solution to work with a somewhat non-strict strategy. We say "somewhat" because in some non-confluent and/or non-terminating classes of TRSs used to model functional logic languages, a notion of laziness is not always precisely defined.

4.1 Non-locality

The notion of a block defining a lexical scope dates back to the dawn of programming languages and continues nowadays. Main early languages incorporating this notion of a block include Lisp and Algol 60. In early Lisp dialects non-local variables were dynamically bound. Dynamic binding is easier to implement, but more difficult to understand and control. For this reason, modern Lisp dialects such as Scheme switched to static binding. Algol and its descendants, e.g., Pascal and Ada, adopted static binding from their beginning. More recently, a notion of scope was proposed for rewrite rules too [20]. This notion is quite different from ours and will be discussed in Section 6.

The *contour model* provides an operational semantics for nested blocks with static binding. This model relies on the fact that all these languages are eager in the sense that the arguments of a function or procedure call are evaluated before the function's invocation. The binding of a variable, consisting of attributes such as value, location, storage size, etc., are kept in the stack frame of a procedure invocation. The stack discipline of function calls ensures that any time a non-local variable needs to be accessed, the stack frame containing the binding of the variable is somewhere down the stack. For static binding two techniques have been developed to fetch the binding of a variable: the *static chain* and the *display*. The *static chain* is based on a chain of pointers whereas the *display* is based on a table. Dynamic binding is generally implemented by a combination of a *dynamic chain* and a symbol table. Both static and dynamic chains thread stack frames, the first one in the order in which functions are textually nested in a program and the second one in the order in which functions are invoked during an execution.

Given this vast body of knowledge about implementing local scopes in programming languages, one wonders if some of the above techniques could be adapted to rewriting. The answer is likely not. All the above techniques are stack based, i.e., the stack frames containing the bindings of variables, as the name suggests, are stored according to a first-in last-out discipline. This discipline works well for eager evaluation since it ensures that when the access to a non-local variable is required, the frame containing the variable's binding is available on the stack. Eager evaluation in a procedural language is similar to an innermost strategy in rewriting. Therefore, we believe that both techniques for accessing non-local variables in a procedural Algol-like language could be adapted to *innermost* rewriting with non-local variables, as well. However, these techniques do not work for some non-innermost strategies. A following example will prove this claim. Thus the above techniques are strategy dependant.

4.2 Scoped TRSs

In this section, we formalize the notion of a scoped TRS. A scoped rewrite rule and a scoped TRS are interdependent concepts defined as follows. A *scoped rewrite rule* is a pair $\langle l \to r, \mathcal{N} \rangle$, where l and r are terms satisfying some conditions presented below and $\mathcal{N}$ is a *scoped TRS*. A *scoped TRS* is a pair $\langle R, \Sigma \rangle$, where R is a set of *scoped rewrite rules*

and Σ is a signature. Sometimes we will omit the adjective "scoped" when talking of scoped TRSs and scoped rewrite rules. If $\langle l \to r, \mathcal{N} \rangle$ is a rule of the TRS $\mathcal{R}$, then both $\mathcal{R}$ and $l \to r$ are said to be *nesting* both $\mathcal{N}$ and any of its rules, and likewise both $\mathcal{N}$ and any of its rules are said to be *nested* in both $l \to r$ and $\mathcal{R}$. Both a TRS and its rewrite rules are said to be *top-level* if they are not nested in any other TRS or rule.

The nesting relation defines a tree-like hierarchy of scoped TRSs. Loosely speaking, the set of the rules nested within a rule constitute a nested scoped TRS. For the purpose of this paper, we assume that the signatures of any TRS in this hierarchy is disjoint from the signatures any other TRS in this hierarchy. For scoped TRSs modeling programs, it is acceptable and convenient to allow symbols with the same spelling in distinct signatures. These symbols are distinct and an early phase of the compilation, known as name resolution, identifies symbols according to the language's scope rules. Likewise, we assume that the sets of variables in the left-hand side of any two distinct rewrite rules are disjoint.

Continuing the definition started earlier, we are now ready to state the conditions on the left-hand and right-hand sides of a rewrite rule $\langle l \to r, \langle R', \Sigma' \rangle \rangle$ of $\langle R, \Sigma \rangle$.

1. l is a non-variable term over Σ.

2. A symbol s *may occur* in r if and only if one of the following conditions hold:

   (a) s is in Σ or

   (b) s is in Σ' or

   (c) s may occur in the right-hand side of the nesting rule, if $l \to r$ is not top-level.

3. A variable v *may occur* in r if and only if one of the following conditions hold:

   (a) v occurs in l or

   (b) v may occur in the right-hand side of the nesting rule, if $l \to r$ is not top-level.

The intuition behind the above conditions follows. Condition 1 is standard for ordinary TRSs. Condition 2 establishes that a symbol s is acceptable in the right-hand side of a rule m if s is in the signature of (a) the TRS to which m belongs or (b) the TRS nested in m or (c) in any TRS directly or indirectly nesting m. Likewise, condition 3 establishes that a variable v is acceptable in the right-hand side of a rule m if (a) v occurs in the left-hand side of m, which is standard for ordinary TRSs or (b) v occurs in the left-hand side of any rule directly or indirectly nesting m.

The definition of a scoped TRS is a conservative extension of an ordinary TRS. An ordinary TRS is a scoped TRS in which nested TRSs are empty, or equivalently, all the symbols are top-level. Referring to Display (3), the rules of `reverse` are the only top-level rules. The second rule of `reverse` is nesting the rules of `appelem`. The right-hand side of this rule contains an occurrence of `appelem` which belongs to the signature of the nested TRS. The right-hand side of the first rule of `appelem` contains an occurrence of a variable defined by the left-hand side of the nesting rule.

4.3 Constructor TRSs

TRSs modeling programs in many declarative programming languages, such as Haskell and Curry, have rewrite rules with left-hand sides characterized by a particular structure referred to as the *constructor discipline* [18]. Our technique relies on this characterization.

A TRS $\langle R, \Sigma \rangle$ is a *constructor TRS* if Σ is a constructor signature and the left-hand sides of the rules of R are patterns. A signature is a *constructor signature* if it is partitioned into a set $\mathcal{C}$ of (*data*) *constructors* and a set $\mathcal{D}$ of (*defined*) *operations*. A *pattern* over a constructor signature $\mathcal{C} \uplus \mathcal{D}$ is a term $f(t_1, \ldots, t_n)$, $n \geqslant 0$, such that f is in $\mathcal{D}$ and every symbol occurring in t_i, $1 \leqslant i \leqslant n$, is either a variable or is in $\mathcal{C}$. In constructor TRSs, a computation leading to a normal form containing occurrences of defined operations is regarded as failed. Display (4) is a constructor TRS (apart from curried application and special infix symbols) in which `reverse` and `appelem` are operations, whereas `[]` and ":" are constructors.

The definition of a scoped TRS proposed earlier is orthogonal to the notion of a constructor TRS. The only effect of combining these concepts is that the left-hand side of any rule becomes a pattern. It also seems sensible to allow constructor symbols only in the top-level signature to ensure that only top-level symbols occur in the result of a computation.

4.4 Scoped Rewriting

In this section, we define the rewrite relation for a scoped TRS. In an ordinary TRS $\langle R, \Sigma \rangle$, every term t in $\textsc{Term}(\Sigma)$ represents a computation. This computation is a sequence of rewrite steps originating from t. By contrast, some terms of a scoped TRS do not represent meaningful computations. For example, referring to the scoped TRS defined by Display (3), the term `appelem` t, where t is some list, does not represent a meaningful computation. If t is not `[]`, some steps are possible, but eventually the computation produces a term with an occurrence of `appelem []`. This is a redex, but no sensible replacement is available for it due to the presence of an uninstantiated non-local variable, `x`, in the right-hand side of the first rule of `appelem`.

As one would expect, to define the rewrite relation in a scoped TRS one has to "carry around" the instantiations of non-local variables. The challenging part is *how* to carry around these instantiations and to access them in a practical way in an implementation of scoped rewriting. This is non-trivial and novel. We already pointed out that the traditional techniques for statically bound, block-structured, eager languages would not work for rewriting with some non-innermost strategies. For our purpose, we associate a substitution to certain positions of a term. These substitutions are used for obtaining the instantiations of non-local variables occurring in a redex replacement when a nested rule is applied. In *constructor* TRSs, which we consider because programs in functional and functional logic languages are modeled by this class, operation symbol occurrences can be tracked along a computation. This works well since only operation-rooted terms can be redexes. Thus, the substitution associated to a position "travels" with its position along a computation. Let $t \to_{p,l \to r,\sigma} u$ be a step and q the occurrence of an operation symbol of t. The following exhaustive and mutually exclusive cases track q in u:

- $p \not\leqslant q$: the position q is not at or below p. In this case,

the replacement of $t|_p$ does not "disturb" the position q. The occurrence of u at q is called a *descendant* [14] of the occurrence of t at q.

- $p \leqslant q$: the position q is at or below p. We distinguish two subcases:
  - $p = q$: in this case, $u|_p$ is the replacement of $t|_p$. The occurrence of u at q is a *trace* [7] of the occurrence of t at q. The trace of (an occurrence of) an operation symbol can be either an operation or a constructor symbol.
  - $p \neq q$: considering only occurrences of operation symbols simplifies this case. Since we consider a *constructor* TRS, q cannot be in any redex pattern of l. Hence, there is an occurrence, say p', of a variable, say x, in l and a possibly empty position p'' such that $p\,p'\,p'' = q$. If x does not occur in r, then the occurrence of q in t is *erased*. Otherwise, let q' be an occurrence of x in r. The replacement of $t|_p$ may relocate and/or multiply the occurrence of t at q, but it does not "disturb" it in any other way. Thus, in this case too, the occurrence of $p\,q'\,p''$ in u is a *descendant* [14] of the occurrence of t at q.

Every occurrence of an operation symbol in u is either a descendant of a similar occurrence in t as discussed above, or is *created* by the reduction of t. The created occurrences are all and only of form $p\,q''$ where q'' is an occurrence of an operation symbol in r.

Thus, along the terms of a computation, occurrences of operation symbols can be created, evolve into descendants, and since we are interested in computations that reach a constructor term, they eventually vanish.

In passing, we observe that it would not be sensible to track occurrences of constructor symbols. Consider the well-known *parallel-or* operation defined by:

```
or True x = x
or x True = x                                   (7)
or False False = False
```

In functional logic languages there is no textual ordering in the rewrite rules defining a function. For example, in Curry, every rule applicable to a term must be non-deterministically applied to ensure the completeness of a computation. The term $t =$ `or True True` rewrites to $u =$ `True`. The term u is a descendant of $t|_1$ when t is rewritten by the second rule, but it is a descendant of $t|_2$ when t is rewritten by the first rule.

An attribute, called *level*, is inductively defined for TRSs, rules, and symbols. The level of a top-level TRS is 0. The level of a TRS nested in a TRS $\mathcal{R}$ is 1 more than the level of $\mathcal{R}$. The levels of a rule and a symbol are the levels of the TRS in which they are defined. In Display (3) the level of `appelem` is 1, whereas the level of any other symbol is 0. Loosely speaking, the level of a TRS and its rules and symbols is the TRS's depth in the tree-like hierarchy implicitly defined by the nesting relation.

Let f be a level-n, $n \geqslant 0$, operation symbol. The *context* of an occurrence of f in a term is a sequence of pairs $\langle l_i \to r_i, \sigma_i \rangle$, for $0 \leqslant i < n$, where $l_i \to r_i$ is a level-i rewrite rule and σ_i is a substitution for the variables of l_i. The context of a term is a generic name for the set of contexts of each occurrence of an operation symbol of the term. By definition, the context of any occurrence of a top-level symbol is an empty sequence of pairs. Of course, not every context is useful for rewriting. Below, we define a notion of usefulness for the context of a term.

Let t be a term. Let p be the position of a redex of t, $l \to r$ a rewrite rule applicable to $t|_p$, and v a non-local variable of r defined by a level-i rewrite rule m. Our assumption of unique identifiers of variables in the left-hand sides guarantees that m is unique as well. We say that the context of t is *useful* if and only if the i-th rule of the context of the root of $t|_p$ is m. The context of any term consisting of top-level symbols is trivially useful. The purpose of a context is to provide instantiations of non-local variables.

For example, a useful context of the term `appelem []` of Display (3) is a sequence consisting of:

$$\langle \texttt{reverse (x:xs) = appelem (reverse xs)}, \{\texttt{x} \mapsto \texttt{3}, \texttt{xs} \mapsto \texttt{[]}\} \rangle$$

We will shortly show that this is a useful context created during the evaluation of `reverse [3]`.

The definition of the rewrite relation for scoped TRSs is fairly more complicated than for ordinary TRSs because a replacement may depend on a redex contracted earlier in a derivation. Information about this redex must be saved for later use and stored in a form easily accessible Let t_0 be a top-level term and $t_0 \to \cdots \to t_k \to \cdots$ a computation. We define both t_k and a useful context of t_k by induction on k as follows.

- Base case. The term t_0 is given. By definition, the context of t_0 is the empty sequence for each occurrence of an operation symbol. This context is trivially useful for top-level symbols.
- Induction case. Let t_k, $k \geqslant 0$, be a term of the computation. Let p be a position of t_k, $l \to r$ a level-n rewrite rule, and σ a substitution such that $\sigma(l) = t_k|_p$. The level of the symbol of t_k at p is n, as well. By the induction hypothesis, a useful context C is associated to this symbol occurrence. C is a sequence of length $n-1$. We define t_{k+1} as $t_k[\eta(r)]_p$ where η is defined as follows. Let v be a variable of r. If v is local, then $\eta(v) = \sigma(v)$. Otherwise, v is defined by a unique rewrite rule R of level m, $m < n$. The context C contains R and an associated substitution σ'. We define $\eta(v) = \sigma'(v)$.

  Now, we define the context of t_{k+1}. Let q' be the occurrence of an operation symbol in t_{k+1}. If q' descends from a position q in t_k, then the context of q' is the context of q. If q' is created by the step, then let m be the level of the root of $t_{k+1}|_{q'}$. Let C' be the context obtained from C by appending the pair $\langle l \to r, \sigma \rangle$ at the end of C and taking the prefix of length $m-1$ of this list. By the induction hypothesis that C is useful for q, it is simple to verify that C' is useful for q'.

By way of example, we show how to obtain the context of `appelem []` discussed earlier, according to the definition. Let `reverse [3]` be the initial term. The reduction,

$$\texttt{reverse [3]} \to \texttt{appelem (reverse [])}$$

associates the context,

$$\langle \texttt{reverse (x:xs) = appelem (reverse xs)}, \{\texttt{x} \mapsto \texttt{3}, \texttt{xs} \mapsto \texttt{[]}\} \rangle$$

to the occurrence of `appelem`. The inner reduction,

$$\texttt{reverse [] } \rightarrow \texttt{ []}$$

transfers the above context to `appelem []` and gives the desired result.

4.5 Abstract Implementation Extension

Our abstract implementation of rewriting with non-local variables is obtained from the abstract implementation of ordinary rewriting in Section 3 with a few modifications. We had to add an attribute to symbols, a component to both variable and non-variable terms, and to slightly modify the *replace* function. These are exactly the same changes we applied to a narrowing machine [3] for the compiler/interpreter which prompted our research.

We assume that the *level* attribute defined earlier is available for each symbol, including variables. The level of a variable v is simply the level of the rewrite rule whose left-hand side contains an occurrence of v. As we will see shortly, the level of a variable is explicitly stored in its representation. For a program, the level of a symbol is easily computed by the front end of a compiler.

A component, called *context*, was added to non-variable terms. If $\langle R, \sigma \rangle$ is a pair in the context of a term, all that is needed for the implementation described below is only $\sigma(R)$, i.e., the redex to which R was applied. Thus, the context of a term is either another term, which occurred earlier in a computation, or a distinguished value, denoted by $\perp$, which stands for "non-existent." Since the context of a term is a term itself, for any term t_0 the context function defines a finite or infinite sequence of terms, say $t_1, t_2, \ldots$ such that, for all $i \geqslant 0$, t_{i+1} is the context of t_i. It will be easy to verify that the sequences of contexts that we create are always finite because they have the following property: if the level of the root of t_i is n, then the level of the root of t_{i+1} is $n-1$.

A context is useful only for operation-rooted terms, but in the following discussion we disregard this distinction for the sake of simplicity. We remarked earlier that it is sensible to define data constructors at the top-level only, thus the context of a constructor-rooted term would be trivial in any case.

We extend the abstract definition of a non-variable term in (5) as follows, where the first argument of a *Term* is its context.

$$\begin{array}{lcll} \mathit{root}(\mathit{Term}(c, r, a_1, \ldots, a_n)) & = & r & n \geqslant 0 \\ \mathit{arg}(i, \mathit{Term}(c, r, a_1, \ldots, a_n)) & = & a_i & 1 \leqslant i \leqslant n \\ \mathit{cont}(\mathit{Term}(c, r, a_1, \ldots, a_n)) & = & c & n \geqslant 0 \end{array} \quad (8)$$

Furthermore, we overload the function *cont* to traverse a chain of contexts:

$$\mathit{cont}(i, t) = \begin{cases} t & \text{if } i = 0, \\ \mathit{cont}(i-1, \mathit{cont}(t)) & \text{if } i > 0 \text{ and } t \neq \perp. \end{cases}$$

The above definition generalizes the notion of context of a term t to include t itself, i.e., the 0-th context of t is t. This convention is convenient when t is a redex, i.e, an instance of l for some rewrite rule $l \rightarrow r$. In this case, any variable in r is bound to a subterm of $\mathit{cont}(i, t)$, for some $i \geqslant 0$.

Since both the left-hand and right-hand side of a rewrite rule are terms they also have a context. These contexts are never used in our abstract implementation. Thus, it could seem sensible to define two kinds of terms, one without contexts for rewrite rules and one with contexts for rewriting computations with non-local variables. We disregard this optimization in this paper. In many practical implementations of functional logic programming languages that provide narrowing, redexes are found by automata which are almost universally based on definitional trees [1]. These automata are obtained from the rewrite rules, but the rules themselves do not play any explicit role. Thus, whether to include or omit contexts in the rules becomes insignificant.

The *level* of a variable was added to its representation. We extend the abstract definition of a variable term in (6) as follows (the functions *first* and *rest* are unchanged):

$$\begin{array}{lcll} \mathit{occur}(\mathit{Var}(l, p_1, \ldots, p_n)) & = & (p_1, \ldots, p_n) & n \geqslant 1 \\ \mathit{level}(\mathit{Var}(l, p_1, \ldots, p_n)) & = & l & n \geqslant 1 \end{array} \quad (9)$$

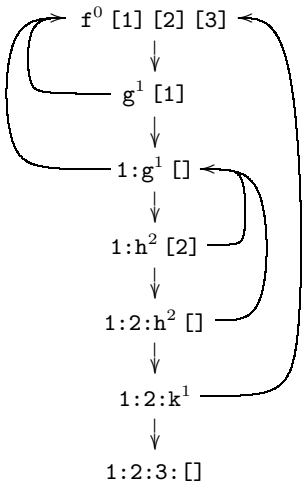
Now, we are ready to present the extended version of *replace*. There are only two small differences with respect to the original version: the argument of *subterm* is a context of the redex in the generalized sense of $\mathit{cont}(\cdot, \cdot)$, rather than the redex itself, and the construction of a term requires an additional argument, a context of the redex in this case, too.

```
Term replace(Term t, Term r) {
  Term subterm(Term t, Occurrence o) {
    ... as before ...
  }
  if isVariable(r) {
    int d = level(root(t)) - level(r);
    return subterm(cont(d,t), occur(r));
  } else {
    int n = arity(root(r));
    int d = level(root(t)) - level(root(r)) + 1;
    return Term(cont(d,t),root(r),replace(t,arg(1,r)), ···,
                                  replace(t,arg(n,r)));
  }
}
```

For example, consider the following program whose purpose is only to show our technique.

```
f x y z = g x
  where g [] = h y
          where h [] = k
                h (u:us) = u : (h us)
        g (v:vs) = v : (g vs)
        k = z
```

We trace the computation of `f [1] [2] [3]`. To ease understanding, we label each occurrence of an operation symbol with its level. Straight arrows pointing down denote rewrite steps. Curved arrows link an operation-rooted term to its context, another operation-rooted term.



Every term of this computation has a single occurrence of an operation symbol, hence both redexes and contexts are easily and uniquely identified.

According to the definition, the context of `f` in the initial term is the empty sequence, and it is not shown.

The replacement of `f [1] [2] [3]`, `g [1]`, is rooted by a level-1 symbol. Its context is rooted by a level-0 symbol. The replacement of `g [1]` contains the subterm `g []`. This subterm has the same context as `g [1]`.

The replacement of `g []`, `h [2]`, is built by accessing a non-local variable. Since the term `h [2]` is rooted by a level-2 symbol, its context is a term rooted by a level-1 symbol.

A variation of this example shows that a stack policy to manage contexts is not adequate for every rewrite strategy. Consider the pair (`f [1] [2] [3]`, `f [4] [5] [6]`) and the rewrite sequence that alternates steps of each component of the pair. The level-0 context of either component must be accessed well after the level-0 context of the other component has been created. Thus, for this strategy, the creation of a context at a given level cannot eliminate or hide the creation of another context at the same level.

4.6 Correctness

In this section we address the correctness of our implementations. Since our implementation language is informal, our discussion is informal too. It is presented only to ease understanding and to increase the confidence in a new algorithm. We assume that a TRS is left-linear and follows the constructor discipline. For scoped TRSs, we assume that the initial term of a computation is made up of top-level symbols each of which has an empty, hence useful, context.

The correctness of the function *match* is expressed as follows: If t is a term and l is a linear term, then there exists a substitution σ such that $\sigma(l) = t$ if and only if $match(t, l)$ is true. The proof is by structural induction on l.

The correctness of the function *replace* for both ordinary and scoped TRSs is expressed as follows: If $l \to r$ is a left-linear rewrite rule, t is a term and σ is a substitution such that $\sigma(l) = t$ then $replace(t, r) = \sigma(r)$. Preliminarily, we observe that if v is a variable of l then $subterm(t, occur(v)) = \sigma(v)$. The proof is by induction on the length of $occur(v)$. The proof of the correctness of *replace* for ordinary TRSs is by structural induction on r.

The extension to scoped TRSs includes two additional computations: substituting local and non-local variables and constructing the context of created occurrences of operation symbols. We assume that the context of each occurrence of an operation symbol of t is useful for the proof of correctness and we prove that the context of each occurrence of an operation symbol of $replace(t, r)$ is useful.

The first computation is reduced to substituting local variables, in fact the same function *subterm* is evaluated, by retrieving the appropriate context. The second computation is reduced to setting the context component of a term with another term which is either the reduct or a context of the reduct. Both computations rely on the function *cont* for retrieving the desired context. The proof is by case analysis on the definition of the function *cont*.

4.7 Narrowing

Narrowing is a generalization of rewriting. A narrowing step differs from a rewriting step in that a non-ground term may be instantiated to create a redex before applying an ordinary rewriting step. Referring to Display (4), the term $t =$ `appelem` t' `X`, where t' is some term and `X` is an uninstantiated variable, cannot be reduced. However, if `X` is instantiated by either `[]` or u`:`u_s, where u and u_s are terms, possibly uninstantiated variables, the corresponding instance of t can be reduced.

Narrowing is the essence of functional logic programming. It seamlessly integrates the two paradigms by supporting the evaluation of functionally nested expressions containing uninstantiated logic variables and thus providing in a functional-like setting both non-determinism and search. In the last decade, a considerable effort has gone into the development of evaluation strategies for narrowing computations. Nowadays, suitable strategies exist, directly or through program transformation, both for the whole class of the conditional constructor TRSs and for several smaller classes of interest for programming [2].

Narrowing strategies generalize rewriting strategies in that given a term t, a strategy computes a set of triples of the form $\langle p, l \to r, \sigma \rangle$, where p is a non-variable position of t, $l \to r$ is a rule and σ is a unifier of $t|_p$ and l. Therefore, $\sigma(t)$ is reducible at p by $l \to r$. When σ is the identity on the variables of t, e.g., if t is ground, a narrowing step becomes a rewriting step.

Most modern narrowing strategies for TRSs modeling functional logic programs are based on *definitional trees* [1]. These strategies, similar to many practical strategies for rewriting, e.g., [14], depend only on the left-hand sides of a TRS's rewrite rules. Therefore, they are independent of the right-hand sides and in particular of the presence of non-local variables. In a typical implementation, the narrowing strategy computes the position, rule and instantiation of a term. Any implementation of rewriting with non-local variables can be employed on the instantiated term.

5. EXPERIMENTAL RESULTS

This section describes some experiments that we conducted to assess the practicality of rewriting with nested rewrite rules. We think that the formalization discussed in Section 4 is important irrespective of the efficiency of its implementation, since it is the only formalization of nested blocks for functional-like computations that includes non-determinism and narrowing. However, it turns out that an implementa-

tion based on the concepts of the abstract implementation of Section 4.5 is practical.

For our benchmarks, we used a virtual machine for functional logic computations under development [3]. The machine is fairly sophisticated. It implements very efficient pattern matching. It executes non-deterministic steps concurrently to ensure the operational completeness of computations. It evaluates only once terms that can be shared across concurrent computations. The instruction set of the machine is relatively simple. It consists of a dozen instructions mostly for pattern matching and term building that efficiently provide the functionality of functions *match* and *replace* presented in Section 3.

The code generators for this machine and other implementations of functional logic languages, e.g., [11], lambda-lifts nested rewrite rules even though this transformation has been validated only empirically for non-deterministic and narrowing computations. We have easy access to the machine source code. Thus we modified both the machine's source code and the compiler's generated bytecode to implement our formalization. We added a single instruction for accessing non-local variables. The other modifications where sparse and simple. We found programs that ran slower and programs that ran faster. For programs that have no nested rules, or were already lambda-lifted, the typical slowdown was about 8%. This is the cost of carrying around a context that is never used. For programs with some nested rules the slowdown was smaller or in some cases there was a speedup.

To understand why a program with nested rules may run faster, consider the following usual formulation of list concatenation:

```
append []     y = y
append (x:xs) y = x:append xs y                    (10)
```

The second parameter is passed untouched through every recursive call only to be returned by the final invocation of `append`. The cost of passing this parameter can be avoided at least partially. The following code defines the same list concatenation of Display (10) using a scoped TRS with nested rules.

```
append x y = appd x
   where appd []     = y                           (11)
         appd (z:zs) = z:appd zs
```

To concatenate two lists, this version makes approximately the same number of calls, but these calls are simpler because only one parameter, instead of two, has to be passed and therefore is more efficient.

For functional languages, the transformation from Display (10) to Display (11) is called *lambda-dropping*, which is somewhat the inverse of lambda-lifting. Lambda-dropping transforms recursive equations into a block structured functional program. It is known [9] that lambda-dropping might improve the efficiency of some computations. We performed some experiments to quantify this claim in the setting of scoped TRSs using the modified virtual machine. Our experimental results suggest that when enough parameters are dropped there is a speedup and that the speedup increases as more parameters are dropped. We ran our experiments using the function f below which has no intuitive meaning, but it allows us to estimate with reasonable accuracy the cost of lambda-lifting/dropping one parameter. The function is intended to recursively call itself decrementing the first parameter at each call until it is zero and then add together the remaining parameters, which are not unchanged in the recursive calls.

$$f(x_0,\ldots,x_N) = \textbf{if } x_0 == 0 \textbf{ then } x_1 + \cdots + x_N \textbf{ else } f(x_0 - 1,\ldots,x_N)$$

We considered two implementations of f, `fd` and `fl`. For a fair comparison, we take into account how a compiler would efficiently compile the "**if** $\cdot$ **then** $\cdot$ **else** $\cdot$" operation and we keep the dropped and lifted implementations as similar as possible.

```
fd x0 ... xN = fdAux (x0 == 0) x0
   where fdAux true  x = x1 + ... + xN
         fdAux false x = fdAux (x == 0) (x - 1)

fl x0 ... xN = flAux (x0 == 0) x0 ... xN
flAux true  x0 ... xN = x1 + ... + xN
flAux false x0 ... xN =
   flAux (x0 == 0) (x0 - 1) x1 ... xN
```

We ran `fd` with the virtual machine modified for scoped TRSs and `fl` with the standard machine. We let N, the number of parameters, be 3, 5, 7, and 10. The benchmarks were run with `x0`, the number of recursive invocations, varying from 0 to 500,000 in increments of 20,000. We found that the execution time increased nearly linearly with `x0`, so we performed a linear regression on the data. To get the speedup of the scoped TRS, we divided the slope obtained for `fl` by that obtained for `fd`. Table 1 shows the speedups we obtained.

N	3	5	7	10
Speedup	1.06	1.16	1.29	1.43

Table 1: Speedup of accessing N arguments of function f, defined in the text, non-locally instead of locally.

We then performed a linear regression on the values of N versus speedup data. The intercept was 0.91 and the slope was 0.05. This suggests that if no parameters were passed to `fd` and `fl` the scoped TRS would have a 9% slow down. It also suggests that for each parameter added there would be an additional 5% speedup in the scoped TRS.

To determine the cost of maintaining the context without obtaining any benefit from it we repeatedly executed `append` and the standard `foldr` (after transforming it into first-order code) with lists from 0 to 300,000 elements in increments of 20,000 on both the scoped and the ordinary implementations of rewriting. Once again we performed a linear regression on the data and measured the speedup as the slope obtained for the ordinary TRS divided by the slope obtained for the scoped TRS. The results are summarized in Table 2.

Function	append	foldr
Speedup	.99	.93

Table 2: Speedup, actually slowdown, of constructing and carrying around a context that is never used for some popular functions.

We did not run any experiments to analyze memory use because this was a difficult task using our implementation, but we can make some pertinent observations. Storing the

context in a term takes as much memory, a pointer, as passing a parameter. Thus, if on average one can trade more than one parameter for the context there will be some memory savings for terms. However, the terms that make up a context may stay around longer before being garbage collected. In general, a scoped TRS will need more heap space than an ordinary TRS since it has to hold on to terms longer.

The context is needed only by functions that access non-local variables. An optimizing compiler could detect some situations in which the context is never used and therefore the compiler could generate code which avoids the cost of both building and carrying around the context, e.g., for functions that have no nested functions and for functions that have nested functions that do not access non-local variables. More aggressive optimizations are also possible. The compiled code of our experiment did not include this or any other optimization.

6. RELATED WORK

To our knowledge, this is the first attempt at formalizing a notion of nested block with static binding for a TRS. Thus, there is no closely related work to compare. A notion of scoped dynamic rewrite rule is proposed in [20]. In that context, "dynamic" means that a rule can be asserted and retracted dynamically, i.e., while a term is being rewritten with a given TRS, some rule may be added to and removed from the TRS. Similar to our work, these dynamic rules may refer to non-local variables, but by contrast to our work the binding of the non-local variables is dynamic rather than static.

Our work has some similarity with the closure conversion transformation. A term in a scoped TRS holds the information of both an ordinary term and the bindings of non-local variables, which is equivalent to the usual environment of a closure conversion. It is interesting to analyze how the environment is built and accessed. The traditional and most straightforward structure for the environment is an array indexed by integers associated to the variables. Our environment is structured in a hierarchical fashion and it is built incrementally. Intuitively, these characteristics should make it very efficient to maintain the environment perhaps at the expense of accessing it. However, the patterns in the rewrite rules of a program are seldom deep, thus the binding of a non-local variable can be retrieved from the environment with only a few operations. Therefore, when seen as a closure conversion our approach seems to offer a good compromise between construction and access.

7. CONCLUSION

We have introduced a notion of a TRS in which some rules can be nested within another rule. Nesting of rewrite rules is intended to define a local scope with static binding of variables. This formalism provides a rigorous semantics for nested definitions in functional logic languages. The syntax of this construct is identical in functional languages and functional logic languages. The semantics in functional languages, nested function definitions in the lambda calculus, is inadequate in functional logic languages because functional logic languages support non-determinism and are executed by narrowing.

We have formally defined the rewrite relation for scoped TRSs. Our definition is a conservative extension of the ordinary rewrite relation. We have presented an abstract implementation of ordinary rewriting and we have extended it to scoped rewriting with simple adaptations. We have presented informal proofs of the correctness of our implementations. The principles behind our implementation of scoped TRSs are quite different from those of block-structured imperative languages because the concepts of rewriting and normalization are defined without involving a strategy. Thus, our implementation must work for any strategy.

Since our notion of scoped TRS is interesting for functional logic programming, we have performed benchmarks for scoped rewriting with a virtual machine for narrowing computations. We have found that our formalism can be directly implemented with a modest effort and that its efficiency is comparable with available alternatives.

8. REFERENCES

[1] S. Antoy. Definitional trees. In H. Kirchner and G. Levi, editors, *Proc. of the 3rd International Conference on Algebraic and Logic Programming*, number 632 in Lecture Notes in Computer Science, pages 143–157, Volterra, Italy, September 1992. Springer-Verlag.

[2] S. Antoy. Evaluation strategies for functional logic programming. In B. Gramlich and S. Lucas, editors, *Electronic Notes in Theoretical Computer Science*, volume 57. Elsevier, 2001.

[3] S. Antoy, M. Hanus, A. Tolmach, and J. Liu. Architecture of a virtual machine for functional logic computations. Preliminary draft, 2003. Available at http://www.cs.pdx.edu/~antoy/.

[4] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[5] M. Bezem, J. W. Klop, and R. de Vrijer (eds.). *Term Rewriting Systems*. Cambridge University Press, 2003.

[6] P. Borovansky, C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen. An overview of ELAN. In C. Kirchner and H. Kirchner, editors, *Electronic Notes in Theoretical Computer Science*, volume 15. Elsevier, 2000.

[7] G. Boudol. Computational semantics of term rewriting systems. In M. Nivat and J. C. Reynolds, editors, *Algebraic methods in semantics*, chapter 5, pages 169–236. Cambridge University Press, Cambridge, UK, 1985.

[8] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. The Maude 2.0 system. In R. Nieuwenhuis, editor, *Rewriting Techniques and Applications (RTA 2003)*, number 2706 in Lecture Notes in Computer Science, pages 76–87. Springer-Verlag, June 2003.

[9] O. Danvy and U. P. Schultz. Lambda-dropping: transforming recursive equations into programs with block structure. *Theoretical Computer Science*, 248(1–2):243–287, 2000.

[10] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, pages 243–320. Elsevier, 1990.

[11] M. Hanus, S. Antoy, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, and F. Steiner. PAKCS: The Portland Aachen Kiel Curry System, 2003.

[12] M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.7.2). Available at http://www.informatik.uni-kiel.de/~curry, 2002.

[13] C. M. Hoffmann and M. J. O'Donnell. Pattern matching in trees. *JACM*, 29:68–95, 1982.

[14] G. Huet and J.-J. Lévy. Computations in orthogonal term rewriting systems. In J.-L. Lassez and G. Plotkin, editors, *Computational logic: essays in honour of Alan Robinson*, pages 395–443. MIT Press, Cambridge, MA, 1991.

[15] T. Johnsson. Lambda lifting: Transforming programs to recursive equations. In *Second International Conference on Functional Programming Languages and Computer Architecture*, pages 190–203, New York, September 1985. Springer-Verlag.

[16] J. Klop. Term rewriting systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume II. Oxford University Press, 1992.

[17] F. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proc. of RTA'99*, pages 244–247. Springer LNCS 1631, 1999.

[18] M. J. O'Donnell. *Computing in Systems Described by Equations*. Springer LNCS 58, 1977.

[19] M. van den Brand, A. van Deursen, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, and J. Visser. The ASF+SDF meta-environment: A component-based language development environment. In *Proceedings of the 10th International Conference on Compiler Construction*, Lecture Notes in Computer Science, pages 365–370. Springer-Verlag, 2001.

[20] E. Visser. Scoped dynamic rewrite rules. In M. van den Brand and R. Verma, editors, *Electronic Notes in Theoretical Computer Science*, volume 59. Elsevier, 2001.